Computations on Arbitrary Surfaces in FDTD Space

Zdzisław Meglicki[a], Boyana Norris[b], Stephen Gray[c]

[a] *Indiana University, Office of the Vice President for Information Technology*
[b] *Argonne National Laboratory, Mathematics and Computer Science Division*
[c] *Argonne National Laboratory, Center for Nanoscale Materials*

Abstract

We describe an algorithm and data structures to carry out auxiliary computations on arbitrary surfaces cutting through the computational domain of FDTD, and their implementation within the Guile process grafted on top of a legacy application. The tools so developed are then used to verify the code by comparing numerical and analytical solutions for harmonic wave scattering on a dielectric ball.

Keywords:
computational electrodynamics, finite-difference time-domain method, fluxes, fourier analysis, near-to-far field transformation, surfaces, virtual measurements, code grafting, object-oriented programming, component-based software engineering
2000 MSC: 65Mxx, 65Txx, 65Zxx, 58Cxx, 58Zxx, 68Uxx, 68Nxx, 7801, 7804, 8508

1. Introduction

Seasoned programmers are familiar with GNU Emacs [1], which is commonly thought of as an editor but is far more. The program's small, compiled C-language core is extended by a sizable library of numerous utilities, mostly scripted in Emacs Lisp, that provide context sensitive editor modes, calculators, news and mail readers, *etc.* The methodology of extending existing applications in this way, or of designing them from the beginning to be so extensible, is well known to software engineers and has been utilized in graphics software as well as in some CAD/CAM tools and problem-solving environments. Application-specific languages have frequently been designed and implemented for this purpose, at some expense in terms of the programmer's labor. A viable alternative is to employ a popular, general-purpose extension language such as Python, Tcl, or GNU Guile [2].

On the other hand, applications developed by programming scientists and engineers seldom make use of the technique, even though it is relatively easy to master, the primary obstacle being the often-intricate interaction between the extension and the core languages and processes or threads they describe. Nevertheless, solid benefits can be derived from the methodology: new and powerful functionality can be added to existing codes without having to delve into their internal logic.

We have found this methodology especially helpful in verifying partial differential equation (PDE) codes and algorithms. To fully test a PDE code, one has to explore in detail its behavior for a wide range of initial and boundary conditions and for various media properties and distributions within the computational domain. Yet, existing codes may not have sufficient flexibility in their input and output routines to allow for needed experimentation and testing. Grafting an extension language onto the code overcomes this deficiency. So enhanced, the code can be extended beyond its original mode of operation and domain of applicability, even by the end users themselves—as has happened with Emacs over the years.

In this context, of specific interest to us is the finite difference time domain (FDTD) method of computational electrodynamics, originally invented by Yee [3] and popularized by Taflove and Hagness [4], although the programming methodology discussed here applies more broadly and would benefit related codes such as NekCEM [5].

FDTD is a second-order accurate, explicit numerical method for solving Maxwell's equations. It mounts electric fields on cell edges and magnetic

fields on cell faces (or vice versa) and advances them one over the other in a leapfrog fashion, in effect ensuring automatic satisfaction of the divergence equations. The method is relatively straightforward to code, yet it has proven remarkably powerful and flexible in its ability to simulate systems and phenomena from nanophotonics through microwave circuit design to scattering of light on cometary grains and interaction of cell phones with brain tissue.

What makes FDTD so effective is not so much its basic time step as the wealth of auxiliary techniques, among them methods for signal absorption on the boundary of the computational domain [6, 7], the division of the domain into total and scattered field regions that allows for injection of plane wave signals [8, 9], methods for media description with the help of auxiliary differential equations [10, 11, 12], and methods for Fourier analysis. These have been contributed and communally refined by its users over the decades since the method's conception.

As our own work called for simulations of increasing complexity [13, 14], in terms of both the materials investigated and their distributions, as well as more thorough code and algorithm verification, we found that conventional methods of input specification and code control were no longer adequate. A solution was to equip the codes in a powerful extension language that would let us prototype additional utilities without having to gut the codes themselves—thus protecting and building on the existing investment.

Among the utilities developed, a group of some importance comprises structures and methods for computations on surfaces: surface maps, fluxes, frequency and polarization filters, and far-field measurements. Together they provide us with a framework for virtual measurements that can be carried out while the computation is unfolding, or afterwards, on data from the application. Most commonly such operations are performed on rectangular planes coincident with the computational grid. Although efficient, this approach is needlessly restrictive. Situations may arise in complex simulations such that fluxes have to be computed through variously shaped surfaces, inserted in strategic locations. To this effect we have developed a flexible apparatus that lets us specify arbitrary surfaces cutting through the computational domain, on which arbitrary operations on the computed fields can be carried out.

In this paper we discuss the architecture of our solution and illustrate it with simple examples derived from the verification of one of our FDTD codes.

2. Code Extension

We have implemented the ideas outlined in the introduction by grafting the GNU extension language, Guile, onto a conventional C++/Fortran FDTD code—called FORMS and built on top of the Chombo toolkit [15]—that computes, in three dimensions, the scattering of arbitrary electromagnetic signals incident from arbitrary directions on arbitrarily distributed media with arbitrary optical properties, all within both total and scattered field regions, surrounded by UPML boundaries [6].

Guile mostly overlaps with Scheme, and its stand-alone version can be used to develop and test code components interactively. When grafted onto an application, it runs in a separate thread, which the application can communicate with to obtain or modify data and to defer evaluations of Scheme forms to. Our choice of Guile as an extension language for the project—Python [16, 17] would be an obvious alternative—was predicated on a similar choice made by the authors of the MEEP FDTD package, which is familiar to electrical and electronic engineers [18]. Although the resulting architecture of our code is somewhat different, the goal was to provide a similar scripting environment to the users.

At the program's startup, the Guile thread loads an input file that contains specifications for the simulation, including parameter values defined as Scheme bindings and functional (lambda) expressions that define media distribution and properties, incident signals, and the virtual measurements to be carried out.

A C++ class, **SCMParmParse**, patterned after **ParmParse** of Chombo, was implemented to facilitate the movement of data between the Chombo layer of the application and the Scheme's binding table. But unlike the latter, it implements the movement of data in *both* directions, as well as providing methods for checking whether the data item in the Scheme space is a procedure, closure, or thunk. Furthermore, it provides methods for fast access to data, for reading and writing, without going through the Scheme binding table, if the corresponding Scheme variable (which is not the same as a binding) is known. The latter is important, as it speeds data transactions significantly.

Functions that are going to be invoked frequently (for example, functions that define material properties) may be coded in C or Fortran, hand-optimized, compiled, and loaded into the Guile thread dynamically as external subroutines.

This architecture, while not the most efficient computationally, gives us flexibility that suits code verification and numerical experimentation. Material properties, distribution, signals, and measurements, can be defined as simple Scheme functions, even pretested in the stand-alone Guile environment, then passed to the program as data written in the input file. Once evaluated in this context and found promising, they can be recoded in C or Fortran for semi-production runs.

For this strategy to work, we had to implement a small number of external Scheme subroutines in the C++/Fortran core of the original program, to provide the Guile thread with read/write access to Chombo arrays and, for parallel execution, with utilities for elementary exchange of data between MPI processes.

The routines that write data on Chombo arrays have *drawing* semantics. They let program users draw boxes (aligned with the grid principal directions), balls, and arbitrarily oriented and shaped ellipsoids, cylinders, cones, disks, parallelepipeds, tori, and lenses (intersections of two overlapping balls) in the program's three-dimensional computational domain—with arbitrary *colors.* The colors are simply integer labels, which the users may then associate with different material properties within their input. The figures may overlap and may be drawn with *color zero*, which wipes out any previously drawn color. Negative colors are reserved for the program's internal use and mark UPML boundaries and the scattered field region. A provision is included for internal definitions of most commonly used materials, such as dielectrics, Drude metals, and single resonance Lorentz media within the core, to be associated with other negative colors.

Only one routine, called **interpolate**, reads data from Chombo arrays and can be used in virtual measurements. It uses volume weighting to interpolate the data for E_i, D_i, H_i, or B_i, $i = x, y, z$, at arbitrary points—not necessarily coincident with the grid nodes—in the computational domain. It knows about the interpolated fields' placements (cell center, edge, or face). A Scheme-level overloading lets the user interpolate for the full vector, $\boldsymbol{E}$, $\boldsymbol{D}$, $\boldsymbol{H}$, or $\boldsymbol{B}$, in one call.

Routines for parallel computation return the total number of processes in the pool and a calling process number, and then perform gather, broadcast, and barrier operations. In the program's core, Chombo takes care of parallel computation transparently. Hence these procedures are for use only in the Guile thread. When invoked, the Guile thread passes the data to the Chombo core of an MPI process, which transmits it to other MPI processes, to be

picked up by Guile threads associated with them.

Functions for computations on surfaces are built on top of several Guile libraries that extend the functionality of the core code. The most basic are libraries that define important physics and mathematics constants and three-dimensional vector calculus—the latter based on GOOPS, which is the object-oriented extension to Guile, similar to CLOS, the Common Lisp Object System [19]. This is then extended by utilities for signal definitions that include various chirps and windows, as well as utilities that define media types and provide lightweight solvers for auxiliary differential equations. Bessel functions and Legendre polynomials from the GNU Scientific Library are wrapped for dynamic loading into Guile, and a library based on them computes the Mie scattering solution for the spherical case and for the arbitrarily polarized incident harmonic wave propagating in the z direction [20, 21]. It is then used to verify the code itself and test the accuracy of numerical solutions.

3. Surface Class

The virtual measurements library defines a **surface** as a class with slots for the following:

1. A parameteric equation that defines the surface.
2. Initial and final values for the parameters.
3. A restriction condition to be additionally imposed on the parameters.
4. Parameter increments.
5. A list of plaquetes covering the surface.
6. Other data, to be specified dynamically by the user or utilities provided.

A **plaquette** is also defined as a class with the following slots:

1. The plaquette's center, which is a three-dimensional vector.
2. The parameters of the plaquette's center.
3. The plaquette's surface element, which is also a three-dimensional vector, that corresponds to $\boldsymbol{n}\,\mathrm{d}^2S$, where $\boldsymbol{n}$ is a unit vector perpendicular to the plaquette, and the plaquette's area is evaluated by using the Brahmagupta-Bretschneider-Coolidge formula [22]

$$\mathrm{d}^2S = \sqrt{(s-a)(s-b)(s-c)(s-d) - \frac{1}{4}(ac+bd-pq)(ac+bd-pq)}, \tag{1}$$

where a, b, c, and d are the lengths of the four sides of d^2S, s is the semiperimeter of d^2S, and p and q are the lengths of the diagonals of d^2S. Although this formula is costly, the computation needs to be carried out once only for a given surface.

4. Other data, to be specified dynamically by the user or utilities provided.

The reason we resort to the Brahmagupta-Bretschneider-Coolidge formula while computing surface elements is that using a simpler expression such as $|\mathbf{e}_u \times \mathbf{e}_v|$, where u and v are surface parameters, can produce a systematic error while computing fluxes. Let us consider, for example, a sphere. Here $|r \sin\theta \mathbf{d}\theta \times r\mathbf{d}\phi|$ underestimates the plaquette's area in the northern hemisphere, for finite $\mathbf{d}\theta$ and $\mathbf{d}\phi$, because the meridians diverge from the north pole. In the southern hemisphere, the same expression overestimates the area, because the meridians converge toward the south pole. Whereas the latter compensates the former exactly when computing the surface area, the compensation will not work when computing a flux of, for example, a constant field, $\boldsymbol{E} = E\boldsymbol{e}_z$. Hence we need a more general formula that evaluates accurately the area of an arbitrary quadrilateral.

To prepare a surface for computations, the user or a utility must create an object of the **surface** class by providing the parametric equation, the initial and final values for the parameters, parameter increments, and, sometimes, a restriction condition—for example, a disk covered with a rectangular grid requires one. To cover the surface with the plaquettes, the user then invokes a method called **iterator**, which generates the list of plaquettes and places it in the **plaquettes** slot of the **surface** object.

To assist the user, the library provides methods for generation of commonly used surfaces, such as hemispheres, spheres, planes, disks, and spherical caps, the last parametrized by a projection onto the plane tangent to the cap's apex from the opposite point of the sphere. Spherical caps can be used to cover the polar regions of the sphere, so as to avoid singularities of the spherical coordinate system.

On this foundation we build utilities for construction of field maps on surfaces, computation of fluxes through surfaces, Fourier analysis of fields on surfaces, and far-field measurements.

4. Field Maps

Field maps are a commonly employed utility for field visualization. When combined with maps produced using analytical solutions, they provide a

simple tool for code verification. An example is presented in Figure 1.

Here we simulated, on a 190^3 grid,[1] scattering in vacuum of a monochromatic x-polarized wave of amplitude 1 and length $\lambda = 60$, moving in the z-direction, on a glass ball ($\epsilon_{SiO_2} = 3.8$) of radius $r_{glass} = 27$. The units of length are arbitrary because of the conformal invariance of Maxwell electrodynamics: in all computations we hide ϵ_0 and μ_0 into the fields themselves and assume $c = 1$. Of course, proportions such as r_{glass}/λ are not arbitrary, and, together with ϵ_{SiO_2} (which is dimensionless), correspond to *a class* of specific physical systems.

In the figure, the two panels on the left show numerically computed values of E_x at some instant $t = t_0$, in two planes, $x = x_c$ at the top and $y = y_c$ at the bottom, where $\boldsymbol{r}_c = [x_c, y_c, z_c]$ is the ball's center. The abscissa corresponds to the z direction. The two panels on the right show analytically computed values of E_x, using the Mie solution, in the same two planes and at the same instant. The color and contour scales are the same in all four panels. The comparison shows the numerical solution to be good, although minor differences exist: there is too much backscattering in the numerical solution, for example, which is caused by the staircasing of the ball's surface: it becomes a disk rather than a cap in the vicinity of the pole facing the incident wave.

Using the library utilities provided, one can define more surfaces and compute field maps on them, in order to explore as much of the whole three-dimensional domain as is reasonable and in sufficient detail.

Figure 2 in Section 6 shows a map of $\boldsymbol{n}_p \cdot \bar{\boldsymbol{P}}(\omega, \boldsymbol{r}_p)$, where $\bar{\boldsymbol{P}}(\omega, \boldsymbol{r}_p)$ is defined by equation (6), on a sphere of radius 75 co-centric with the glass ball, for the same simulation.

The computation of the map proceeds by running down the list of plaquettes, and invoking **interpolate** at $\boldsymbol{r}_p$, for each plaquette, for a field of interest. The interpolated field can be then written to a file or saved in the plaquette's **other−data** slot for further processing, if required.

5. Simple Fluxes

With the **surface** object instantiated and initialized, flux computations can be carried out easily because most of the work is already done by the

[1]The grid included UPML boundaries and scattered and total field regions. The total field region itself was covered by a smaller 84^3 grid.

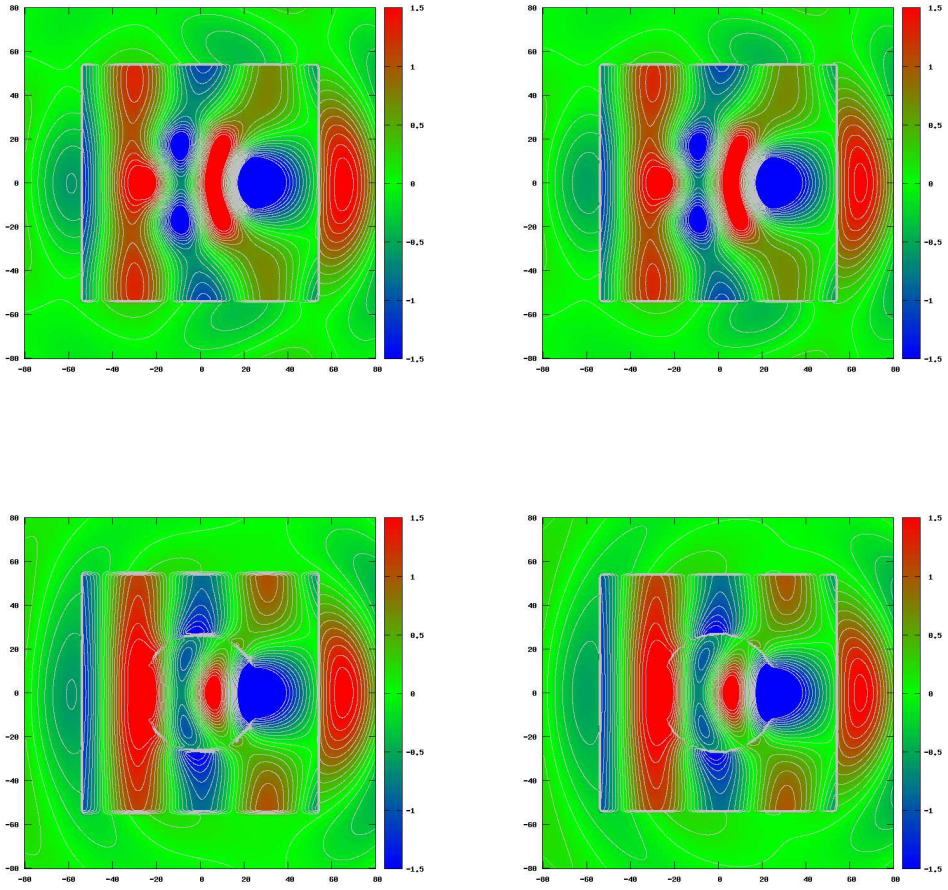

Figure 1: Comparison between numerically and analytically generated fields—total within the total field box and scattered outside—for scattering of an incident x-polarized harmonic signal on a glass ball. E_x values have been collected on two perpendicular slices that cut through the center of the ball, $x = x_c$ at the top and $y = y_c$ at the bottom. The two panels on the left show a numerically generated solution; the two panels on the right show an analytically generated solution for the same instant.

iterator. For example, to evaluate the surface area, $\mathcal{A}$, one would run down the list of plaquettes accumulating their norms: $\mathcal{A} = \int_S |\boldsymbol{n}\,\mathrm{d}^2 S|$. This provides us with a useful check for whether the surface is correctly defined and whether the plaquette covering is sufficiently fine.

Since one can think of the surface area as a flux of $\boldsymbol{n}$ through S, this procedure is at the same time a prototype for more general flux computations: in this case the field in question, say, $\boldsymbol{E}$ or $\boldsymbol{H}$, may be extracted from the Chombo data with **interpolate** for the plaquette's centre, before evaluating its scalar product with the plaquette's surface element and the accumulation step. The actual flux computation method is overloaded so that it computes a normal flux for a single vector field and a cross-product flux, for example, $\int_S (\boldsymbol{E} \times \boldsymbol{H}) \cdot \boldsymbol{n}\,\mathrm{d}^2 S$, if two vector fields are specified in the arguments list.

6. Fourier Analysis

The first step in the Fourier analysis is to specify frequencies for which Fourier accumulations should be carried out. A list of frequencies may be specified manually or by calling a simple function that generates the required frequencies automatically, the latter useful if full spectral analysis is planned.

A predefined surface object must be then prepared for Fourier operations. It is first covered with plaquettes, if this hasn't been done yet. Next, a list is assembled, the first element of which is the field label, and the second element of which is a list of null three-dimensional vectors of length equal to the number of frequencies. The list is then placed in the **other−data** slot of each plaquette.

If the surface is to be prepared for Fourier operations on more than one vector field, the first element of the list is a list of field labels, and the second element is a list of lists of null vectors. The lists attached to each plaquette will be then used to accumulate the Fourier transforms.

For example, let a given plaquette's central point be $\boldsymbol{r}_p$, let the frequencies list be $(\omega_1, \omega_2, \ldots, \omega_{n_f})$, and let us suppose we intend to compute $\hat{\boldsymbol{E}}(\omega_i, \boldsymbol{r}_p)$ and $\hat{\boldsymbol{H}}(\omega_i, \boldsymbol{r}_p)$ for each plaquette, $p = 1, 2, \ldots, n_p$. Then the list of lists in

the **other−data** slot of plaquette p will be

$$\begin{aligned}\Big(\quad & (\text{“E”} \quad , \quad \text{“H”}), \\ \Big(& \big(\hat{\boldsymbol{E}}(\omega_1, \boldsymbol{r}_p) \quad , \quad \hat{\boldsymbol{H}}(\omega_1, \boldsymbol{r}_p)\big), \\ & \big(\hat{\boldsymbol{E}}(\omega_2, \boldsymbol{r}_p) \quad , \quad \hat{\boldsymbol{H}}(\omega_2, \boldsymbol{r}_p)\big), \\ & \quad \ldots, \\ & \big(\hat{\boldsymbol{E}}(\omega_{n_f}, \boldsymbol{r}_p) \quad , \quad \hat{\boldsymbol{H}}(\omega_{n_f}, \boldsymbol{r}_p)\big)\Big)\Big) \end{aligned} \tag{2}$$

when the computation completes.

Because the list of frequencies is common to all plaquettes, it is placed in the **other−data** slot of the surface itself, not of the plaquettes.

With the surface so prepared, the Fourier accumulation, $\int_{t_1}^{t_2} \mathcal{F}(t, \boldsymbol{r}_p)\, e^{i\omega_k t}\, \mathrm{d}t$, for the frequencies and fields specified, for example, $\mathcal{F} = \boldsymbol{E}, \boldsymbol{H}$, can commence. A simple method is invoked at every time-step. The only parameter the method uses is the surface object. All other information, including the fields, the frequencies, and the plaquettes, is read from the object, whereas t and $\mathrm{d}t$ are read from simulation constants, which are global to the Chombo program and shared with Guile. The coding in this case is further facilitated by Guile's automatic recognition of complex numbers and transparent overloads of arithmetic operations on them.

A commonly used approach employs both $\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_p)$ and $\hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_p)$ to evaluate the period average of the monochromatic power vector at ω [21].

Assuming $\boldsymbol{E}(t) = \boldsymbol{E}_c \cos\omega t + \boldsymbol{E}_s \sin\omega t$, and introducing a phasor field $\tilde{\boldsymbol{E}} = \boldsymbol{E}_c + i\boldsymbol{E}_s$, and similarly for $\boldsymbol{H}(t)$, one can show that

$$\frac{1}{T}\int_{t_0}^{t_0+T} \boldsymbol{E}(t) \times \boldsymbol{H}(t)\, \mathrm{d}t = \frac{1}{2}\Re\left(\tilde{\boldsymbol{E}} \times \tilde{\boldsymbol{H}}^*\right), \tag{3}$$

where $T = 2\pi/\omega$. If we Fourier-accumulate $\boldsymbol{E}(t)$ over two full periods,

$$\hat{\boldsymbol{E}} = \int_{t_0}^{t_0+2T} \boldsymbol{E}(t)\, e^{i\omega t}\, \mathrm{d}t = T\tilde{\boldsymbol{E}}, \tag{4}$$

and do similarly for $\hat{\boldsymbol{H}}$, then the final formula is

$$\boldsymbol{P}(\omega, \boldsymbol{r}_p) = \frac{1}{2T^2}\Re\left(\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_p) \times \hat{\boldsymbol{H}}^*(\omega, \boldsymbol{r}_p)\right). \tag{5}$$

Two auxiliary methods are provided to make this task easier. One computes the full flux of $\boldsymbol{P}(\omega, \boldsymbol{r}_p)$ through the surface, assuming that the required Fourier transforms have been accumulated (this is checked by the method on invocation). The other dumps the map of $\boldsymbol{n}_p \cdot \boldsymbol{P}(\omega, \boldsymbol{r}_p)$ on a file, in a format suitable for display with Gnuplot. The latter can be then compared with a similarly generated power map that uses analytical solutions for phasors $\tilde{\boldsymbol{E}}(\omega, \boldsymbol{r}_p)$ and $\tilde{\boldsymbol{H}}(\omega, \boldsymbol{r}_p)$, in order to assess the accuracy of the numerical solution.

Figure 2 shows just such a comparison. The computation is as described in Section 4; see Figure 1. Here, virtual measurements were carried out on a fictitious measurement sphere of radius $r_{\text{measure}} = 75$, which was fully enclosed within the scattered field region. We carried out two types of measurements. First, we measured $\boldsymbol{E}(t, \boldsymbol{r}_p)$ and $\boldsymbol{H}(t, \boldsymbol{r}_p)$ for $t \in [t_0, t_0+T]$, where T was the wave's period, and evaluated

$$\bar{\boldsymbol{P}}(\boldsymbol{r}_p) = \frac{1}{T} \int_{t_0}^{t_0+T} \boldsymbol{n}_p \cdot (\boldsymbol{E}(t, \boldsymbol{r}_p) \times \boldsymbol{H}(t, \boldsymbol{r}_p)) \, \mathrm{d}t \tag{6}$$

for each plaquette $p = 1, \ldots, n_p$. Next, we accumulated $\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_0)$ and $\hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_0)$, as per equation (4), and evaluated $\boldsymbol{n}_p \cdot \boldsymbol{P}(\omega, \boldsymbol{r}_p)$ using equation (5), also for each plaquette p. The resulting two maps were identical to between five and eight significant digits, thus testing the accuracy of our Fourier operations. The map for $\bar{\boldsymbol{P}}(\boldsymbol{r}_p)$ is shown in the top panel of Figure 2.

To verify the FDTD code itself for this computation, we used the virtual measurements library methods (which can be invoked in a stand-alone Guile session, as well as from within the Guile extended FDTD code), but we filled the $\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_p)$ and $\hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_p)$ slots on the measurement sphere's plaquettes with analytically derived field values obtained from the Mie solution (see Sections 4 and 2) The corresponding map is shown in the bottom panel of Figure 2.

The resulting comparison is highly sensitive to any breaks in the symmetry, caused, for example, by interactions with the UPML boundaries, since it piles up errors over Fourier and/or power accumulation time. As a matter of fact, it can be used to adjust the UPML parameters, so as to minimize the global error, which is a methodology we adopted in our simulations. In effect we obtained agreement between the numerical computation and the Mie solution to within a relative error of less than 1%, which is within the expected accuracy of the FDTD method for this configuration and resolu-

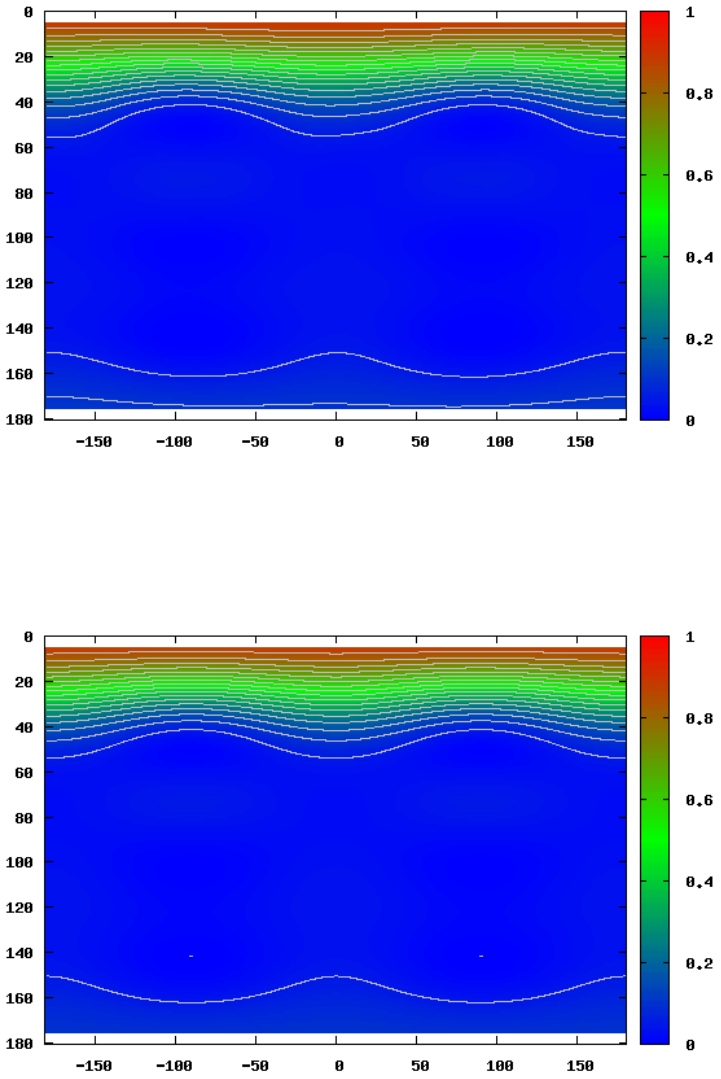

Figure 2: $\boldsymbol{n}_p \cdot \boldsymbol{P}(\omega, \boldsymbol{r}_p)$ on a sphere of radius 75 co-centric with a scattering glass sphere of radius 27, for x-polarized incident monochromatic wave of length 60, moving up in the z-direction in vacuum. Top figure: FDTD computation with FORMS; bottom figure: analytical computation based on the Mie solution. The abscissa corresponds to the azimuth angle, ϕ, from $-180°$ to $+180°$. The ordinate corresponds to the inclination angle, θ, from $0°$ at the top, to $+180°$ at the bottom.

tion.[2] The somewhat stronger backscattering, due to the staircasing of the ball's surface, mentioned in Section 4, can be seen clearly in the upper panel: an extra contour line is visible at the bottom of the panel.

7. The Near-to-Far-Field Transformation

The near-to-far-field transformation is an essential tool that lets us carry out virtual measurements on systems that combine physics at various distance scales. Light scattering on cometary dust is a typical example and is in many ways similar to scattering on nanoparticles. This process can be simulated in the vicinity of the scattering particles, but observations are normally made from a very large distance, compared with the size of the particles, which cannot be captured explicitly in a numerical simulation.

Computational methodology developed by Love and MacDonald in the early 1900s provides us with means to compute the far field if a solution is known on a closed surface S_0 that surrounds the scattering particles [23, 24, 25]. We found a more recent discussion in [4] and especially in [26] helpful.

In short, assuming that $\hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_0)$ and $\hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_0)$ are known for all $\boldsymbol{r}_0 \in S_0$ we first compute the *zero-potentials:*

$$\hat{\boldsymbol{A}}_0\left(S_0, \omega, \boldsymbol{e}_r\right) = \frac{1}{4\pi} \int_{S_0} \boldsymbol{n}_0 \times \hat{\boldsymbol{H}}(\omega, \boldsymbol{r}_0)\, e^{ik\boldsymbol{r}_0 \cdot \boldsymbol{e}_r}\, \mathrm{d}^2 S, \quad \text{and} \tag{7}$$

$$\hat{\boldsymbol{F}}_0\left(S_0, \omega, \boldsymbol{e}_r\right) = -\frac{1}{4\pi} \int_{S_0} \boldsymbol{n}_0 \times \hat{\boldsymbol{E}}(\omega, \boldsymbol{r}_0)\, e^{ik\boldsymbol{r}_0 \cdot \boldsymbol{e}_r}\, \mathrm{d}^2 S, \tag{8}$$

where $\boldsymbol{e}_r$ is a unit vector pointing from the center of the coördinate system, assumed to be enclosed by S_0, toward the (far) observation point, $k = 2\pi/\lambda$ is the wave number, and $\boldsymbol{n}_0\, \mathrm{d}^2 S$ is the S_0 plaquette vector. These are then transformed into two electromagnetic vector potentials at the observation

[2] The FDTD algorithm introduces a slight asymmetry into the computation because of how it samples the fields on cell faces and edges. The main sources of computational error in this case are, apart from the relatively coarse gridding, the grid staircase that covers the surface of the scattering glass ball, imperfect signal injection and subtraction on the total/scattered field boundary, and imperfect absorption of the incident signal by the UPML boundary.

point $\boldsymbol{r} = r\boldsymbol{e}_r$:

$$\hat{\boldsymbol{A}}(\omega, \boldsymbol{r}) = \frac{e^{-ikr}}{r}\hat{\boldsymbol{A}}_0\left(S_0, \omega, \boldsymbol{e}_r\right), \tag{9}$$

$$\hat{\boldsymbol{F}}(\omega, \boldsymbol{r}) = \frac{e^{-ikr}}{r}\hat{\boldsymbol{F}}_0\left(S_0, \omega, \boldsymbol{e}_r\right). \tag{10}$$

It is advantageous to compute $\hat{\boldsymbol{A}}_0$ and $\hat{\boldsymbol{F}}_0$ separately, since these can be reused for the whole ray in the $\boldsymbol{e}_r$ direction, but they vary with the direction itself.

The final step computes the electromagnetic fields transforms at the observation point itself, namely,

$$\hat{\boldsymbol{E}}\left(\omega, \boldsymbol{r}\right) = -ik\hat{\boldsymbol{F}} \times \boldsymbol{e}_r - i\omega\left(\hat{\boldsymbol{A}} - \left(\hat{\boldsymbol{A}} \cdot \boldsymbol{e}_r\right)\boldsymbol{e}_r\right), \tag{11}$$

$$\hat{\boldsymbol{H}}\left(\omega, \boldsymbol{r}\right) = ik\hat{\boldsymbol{A}} \times \boldsymbol{e}_r - i\omega\left(\hat{\boldsymbol{F}} - \left(\hat{\boldsymbol{F}} \cdot \boldsymbol{e}_r\right)\boldsymbol{e}_r\right). \tag{12}$$

The formulae already truncate terms that vanish like $1/r^2$ and faster.

To facilitate these and other operations on far fields, we introduced a **far−field** class with slots for the following:

1. Frequency, ω.
2. Direction, $\boldsymbol{e}_r$.
3. Distance, r.
4. Magnetic field potential, $\hat{\boldsymbol{A}}$.
5. Electric field potential, $\hat{\boldsymbol{F}}$.
6. Electric field, $\hat{\boldsymbol{E}}$.
7. Magnetic field, $\hat{\boldsymbol{H}}$.

We also added simple methods for the *addition* of **far−field** objects, and for their *copy,* because we need to add far-field potentials for surfaces that are covered by multiple patches, as is the case for a sphere, before we can evaluate the fields.

With a closed surface object with plaquettes filled with $\hat{\boldsymbol{E}}(\omega_i, \boldsymbol{r}_p)$ and $\hat{\boldsymbol{H}}(\omega_i, \boldsymbol{r}_p)$ vectors, where $p = 1, \ldots, n_p$, and for a list of frequencies, $\omega_i, i = 1, \ldots, n_f$, as shown by (2), we can now call a method that computes $\hat{\boldsymbol{A}}_0$ and $\hat{\boldsymbol{F}}_0$ for a given direction $\boldsymbol{e}_r$ and for each frequency in the list, and returns a list of **far−field** objects, with the $\hat{\boldsymbol{E}}$ and $\hat{\boldsymbol{H}}$ slots still uninitialized and ordered the same way as the list of frequencies. Next, this list can be converted to

Table 1: Far-field transformation versus exact evaluation for $r = 2000$.

ω	$\boldsymbol{n}$	Mode	$\hat{\boldsymbol{E}}/10^{-4}$	$\hat{\boldsymbol{H}}/10^{-4}$
0.42	$[1,1,0]$	far-field	$[-3.74, 3.74, -0.57]$+i $[9.21, -9.21, -11.24]$	$[-0.40, 0.40, 5.28]$+i $[-7.95, 7.59, -13.03]$
		Mie	$[-3.86, 3.80, -0.39]$+i $[9.12, -9.26, -11.20]$	$[-0.25, 0.30, 5.42]$+i $[-7.93, 7.92, -12.99]$
0.84		far-field	$[-4.02, 4.02, 15.79]$+i $[-6.74, 6.74, -1.29]$	$[11.16, -11.16, 5.69]$+i $[-0.91, 0.91, 9.52]$
		Mie	$[-3.90, 4.04, 15.62]$+i $[-6.96, 6.95, -1.13]$	$[11.07, -11.01, 5.62]$+i $[-0.77, 0.83, 9.84]$
0.42	$[0,1,1]$	far-field	$[27.21, 0, 0]$+i $[-49.38, 0, 0]$	$[0, 19.24, -19.24]$+i $[0, -34.91, 34.91]$
		Mie	$[27.77, 0, 0]$+i $[-49.31, 0, 0]$	$[0, 19.64, -19.62]$+i $[0, -34.93, 34.81]$
0.84		far-field	$[-53.73, 0, 0]$+i $[-36.74, 0, 0]$	$[0, -37.99, 37.99]$+i $[0, -25.98, 25.98]$
		Mie	$[-53.01, 0, 0]$+i $[-37.38, 0, 0]$	$[0, -37.49, 37.48]$+i $[0, -26.43, 26.47]$
0.42	$[1,0,1]$	far-field	$[10.76, 0, -10.76]$+i $[-4.48, 0, 4.48]$	$[0, 15.21, 0]$+i $[0, -6.34, 0]$
		Mie	$[10.78, 0, -10.71]$+i $[-4.64, 0, 4.67]$	$[0, 15.19, 0]$+i $[0, -6.58, 0]$
0.84		far-field	$[-16.79, 0, 16.79]$+i $[-13.81, 0, 13.81]$	$[0, -23.74, 0]$+i $[0, -19.52, 0]$
		Mie	$[-16.81, 0, 16.95]$+i $[-14.22, 0, 14.24]$	$[0, -23.87, 0]$+i $[0, -20.13, 0]$
0.42	$[-1,1,0]$	far-field	$[-3.74, -3.74, 0.57]$+i $[9.21, 9.21, 11.24]$	$[0.40, 0.40, 5.28]$+i $[7.95, 7.95, -13.03]$
		Mie	$[-3.86, -3.80, 0.39]$+i $[9.12, 9.26, 11.20]$	$[0.25, 0.30, 5.42]$+i $[7.93, 7.92, -12.99]$
0.84		far-field	$[-4.02, -4.02, -15.79]$+i $[-6.74, -6.74, 1.29]$	$[-11.16, -11.16, 5.69]$+i $[0.91, 0.91, 9.52]$
		Mie	$[-3.90, -4.04, -15.62]$+i $[-6.96, -6.95, 1.13]$	$[-11.07, -11.01, 5.62]$+i $[0.77, 0.83, 9.84]$
0.42	$[0,-1,1]$	far-field	$[27.21, 0, 0]$+i $[-49.38, 0, 0]$	$[0, 19.24, 19.24]$+i $[0, -34.91, -34.91]$
		Mie	$[27.77, 0, 0]$+i $[-49.31, 0, 0]$	$[0, 19.65, 19.62]$+i $[0, -34.93, -34.81]$
0.84		far-field	$[-53.73, 0, 0]$+i $[-36.75, 0, 0]$	$[0, -37.99, -37.99]$+i $[0, -25.98, -25.98]$
		Mie	$[-53.01, 0, 0]$+i $[-37.39, 0, 0]$	$[0, -37.49, -37.48]$+i $[0, -26.43, -26.45]$
0.42	$[1,0,-1]$	far-field	$[-2.23, 0, -2.23]$+i $[21.68, 0, 21.68]$	$[0, 3.15, 0]$+i $[0, 30.67, 0]$
		Mie	$[-2.54, 0, -2.54]$+i $[21.49, 0, 21.44]$	$[0, 3.59, 0]$+i $[0, 30.35, 0]$
0.84		far-field	$[12.11, 0, 12.11]$+i $[14.82, 0, 14.82]$	$[0, -17.12, 0]$+i $[0, -20.97, 0]$
		Mie	$[11.94, 0, 11.88]$+i $[15.36, 0, 15.35]$	$[0, -16.84, 0]$+i $[0, -21.72, 0]$

a list of **far−field** objects, where $\hat{\boldsymbol{A}}_0 \to \hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{F}}_0 \to \hat{\boldsymbol{F}}$ for a given distance r, by a simple map.[3] Then another map is invoked that converts the result to a list of fully computed **far−field** objects with $\hat{\boldsymbol{E}}(\omega_i, \boldsymbol{r})$ and $\hat{\boldsymbol{H}}(\omega_i, \boldsymbol{r})$, for each $i = 1, \ldots, n_f$. For convenience this chain of operations is wrapped into a single method by overloading, that invokes the necessary steps transparently.

To test the computational apparatus, we invoked the Mie solution for the scattering of two incident waves of $\lambda_1 = 15$ and $\lambda_2 = 5$ on a glass ball ($\epsilon_{\mathrm{SiO}_2} = 3.8$) of radius $r_{\mathrm{glass}} = 10$, and evaluated analytically scattered fields on plaquettes of a sphere of radius $r_{\mathrm{sphere}} = 12$, co-centric with the glass ball, which was covered by three patches—the north cap, the south cap, and the normal spherical patch in between—so as to avoid the coördinate system singularities at the poles. Then we evaluated far-fields, $\hat{\boldsymbol{E}}(\omega_i, \boldsymbol{r})$ and $\hat{\boldsymbol{H}}(\omega_i, \boldsymbol{r})$, for the two frequencies in the list and for $r = 2000$, and for the

[3] A map in Scheme is a function that maps one list onto another.

following six directions:

$$\begin{aligned}
\boldsymbol{e}_1 &= (1,1,0) \\
\boldsymbol{e}_2 &= (0,1,1) \\
\boldsymbol{e}_3 &= (1,0,1) \\
\boldsymbol{e}_4 &= (-1,1,0) \\
\boldsymbol{e}_5 &= (0,-1,0) \\
\boldsymbol{e}_6 &= (1,0,-1)
\end{aligned}$$

Finally, we evaluated the exact fields, using the Mie solution, for $\boldsymbol{r}_k = r\boldsymbol{e}_k, k = 1, \ldots, 6$. The results are shown in Table 1.

The ratios of $2r_{\text{sphere}}/r = 0.012$ and $\lambda_1/r = 0.0075$ are not so small here that no difference would be visible between the Mie solution and the far-field approximation. Nevertheless, we find good agreement (with about 1% error) where the field values are relatively high, and the right order of magnitude and field directions everywhere.

8. Conclusions

High-resolution scientific and engineering simulations in three dimensions produce copious amounts of data, which normally still must be postprocessed to deliver information useful to scientists and engineers. A conventional strategy from the days when supercomputer CPUs were slow and expensive was to dump the simulated data as "bricks-of-bytes" or "bricks-of-floats" and postprocess it on cheaper hardware, for example, workstations using visualization and other tools.

As resolution improved and CPUs got cheaper, however, the equation changed in favor of computation and against moving between computational systems involved and storing huge amounts of data often for long time. A strategy that arose was to coprocess the data on the fly. But this calls for embedding data processing routines in the code itself, which increases its complexity, while not contributing to its flexibility.

The use of code extensions, in combination with powerful scripting languages such as Python and Guile, provides a third way. Here the data-processing routines are external yet can be invoked from within the core supercomputer code to carry out whatever auxiliary computations are required. They can be developed externally, tested within their own interactive environments, and assembled into libraries of utilities, to be invoked not

only from within the core code, but also from within post- and preprocessing codes, were these still to be used. Where computational efficiency is needed, which is not always the case, the extensions can be prototyped and tested in a scripting language, then recoded in a low-level language, C or Fortran, for better performance, to be called from within the extension shell, or even embedded directly into the core code.

A special group of co- or postprocessing utilities concerns virtual measurements, and here we find that data is frequently sampled and otherwise processed on surfaces. The methodology described in this paper illustrates how a surface can be instrumented as an object to allow for its flexible functional definition and then for its use in complex operations on computed data. As the data itself is stored within the surface, on its plaquettes, all necessary information for surface computations is provided in one item, which is easy to pass through pipes of operations.

Application examples, focused on the verification of an FDTD code, showed this methodology at work. Here we used the same Guile tools to collect, assemble and display data produced by the code and the corresponding data obtained from an analytical solution (Mie's in this case), looking at field and power maps on various surfaces, and proceeding eventually to far-field virtual observations: all steps greatly facilitated by the framework developed, and without ever having to touch the core code, other than to provide the basic hooks for the extension environment.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] Glickstein, B., *Writing GNU Emacs Extensions*, O'Reilly Media, 1997.

[2] *Scheme Implementations: Gnu Guile*, Books LLC, 2010.

[3] Yee, K., IEEE Transactions on Antennas and Propagation **14** (1966) 302.

[4] Taflove, A. and Hagness, S. C., *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House, Boston, 3rd edition, 2005.

[5] Min, M., Discontinuous Galerkin Method Based on Quadrilateral Mesh for Maxwell's Equations, in *Proceedings of 2005 IEEE/ACES International Conference on Wireless Communications and Applied Computational Electromagnetics*, pages 719–723, IEEE, 2005.

[6] Gedney, S. D., IEEE Transactions on Antennas and Propagation **44** (1996) 1630.

[7] Sullivan, D. M., IEEE Microwave and Guided Wave Letters **6** (1996) 97.

[8] Mur, G., IEEE Transactions on Electromagnetic Compatibility **23** (1981) 377.

[9] Umashankar, K. R. and Taflove, A., IEEE Transactions on Electromagnetic Compatibility **24** (1982) 397.

[10] Kashiwa, T. and Fukai, I., Microwave and Optics Technology Letters **3** (1990) 203.

[11] Joseph, R. M., Hagness, S. C., and Taflove, A., Optics Letters **16** (1991) 1412.

[12] Okoniewski, M., Mrozowski, M., and Stuchly, M. A., IEEE Microwave and Guided Wave Letters **7** (1997) 121.

[13] Montgomery, J. M. and Gray, S. K., Phys. Rev. B **77** (2008) 125407.

[14] Yao, J. M. et al., Angew. Chem. Int. Ed. **47** (2008) 5013.

[15] Colella, P. et al., Chombo Software Package for AMR Applications—Design Document, Technical report, Applied Numerical Algorithms Group, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 2009, available from `https://seesar.lbl.gov/ANAG/chombo/`.

[16] van Rossum, G. and Drake, F. L., PYTHON *2.6 and C or C++: Extending and Embedding Python*, Number Part 5 in Python Documentation MANUAL, SoHo Books, 2009.

[17] Ong, E. T. et al., Multilingual Interfaces for Coupling in Multiphysics and Multiscale Systems, in *Proceedings of the International Conference on Computational Science, May 27–30, 2007*, Lecture Notes in Computer Science, Berlin, 2007, Springer.

[18] Oskooi, A. F. et al., Computer Physics Communications **181** (2010) 687, available from `http://ab-initio.mit.edu/wiki/index.php/Meep`.

[19] Steele Jr., G. L., *Common Lisp—The Language*, Digital Press, 2nd edition, 1990.

[20] Mie, G., Annalen der Physik, IV **25** (1908) 377.

[21] Bohren, C. F. and Huffman, D. R., *Absorption and Scattering of Light by Small Particles*, Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany, 2nd edition, 2004.

[22] Coolidge, J. L., The American Mathematical Monthly **46** (1939) 345.

[23] Love, A. E. H., Philosophical Transactions, Series A **197** (1901) 1.

[24] MacDonald, H. M., Electric Waves, Cambridge: at the University Press, 1902, an Adams Prize essay in the University of Cambridge.

[25] MacDonald, H. M., Proceedings of the London Mathematical Society **s2-10** (1912) 91.

[26] Stephanson, M. B., A Fast Near- to Far-Field Transform Algorithm, Master's thesis, College of Engineering of the Ohio State University, 2007.